Alberto Paoluzzi and Giorgio Scorzelli

# BIM geometry with Julia Plasm

## Functional language for CAD programming

May 31, 2024

Springer Nature

---

| | | |
|---|---|---|
| 1 | 3 | |
| Pre/Post conds | 4 | $\rightarrow$ 5 |
| Example | 6 | |

# Contents

# Chapter 4
# Geometric models

Julia `Plasm` is the best choice to write symbolic geometric models for Building Information Modeling (BIM) and Computer-Aided Design (CAD). Geometric models specify the physical appearance of Architecture, Engineering, and Construction products at any scale, from structural and envelope components to whole buildings and built environments, and are used for design, tender, contract, and collaboration. We ported the functional language `Plasm` to Julia for better supporting design, model generation, and visualization of geometric objects. In this chapter we introduce the great expressive power of `Plasm` geometric types and parametric functions, as well the simple methods used to build parametric assemblies, where objects itself can be used as actual parameters. We show also that `Plasm` offers a general mechanism (Julia dictionaries) to export models characterized by colors, textures, materials, and so on. `Plasm` can be even embedded in the `Jupiter` platform in order to document the design choices step-by-step in digital notebooks.

## 4.1 Plasm geometric types

Even if Julia does not pretend the user specifies the type of data objects, which are inferred at compile time, it may always be useful to annotate with their type the parameters and the returned value from function applications in order to get faster codes from the Julia compiler. The best reason concerns program documentation, making it easier to understand the Julia's sources.

Let's remember that `Plasm` derives from three founts: (1) the classic `PLaSM` set up on `FL` functional combinators; (2) the porting to Python (object-oriented language), and finally (3) the embedding into Julia (functional and multi-paradigm), after ten years of algebraic research finalized to understand the role of topology in elaborating digital geometric models.

This development defined different data types and user structures, which the current version of the language proudly unifies by scheduling them to different roles and uses.

1. The Hierarchical Polyhedral Complex, now denoted in Julia `Plasm` as the `Hpc` datatype, was characterized by models defined as aggregation of multidimensional convex cells, described only by their vertices and by multidimensional affine matrices.
2. Our research about algebraic topology of geometric design directed us to design the Linear Algebraic Representation, currently the `Lar` datatype, used to work with chain complexes, and able to fully specify the geometry and topology of the *solid objects* under consideration.
3. Finally, a third Julia user-defined `struct`, named `Geo` for Geometry, is being used as container of huge datasets for `Plasm`-coded applications of `BIM` objects and 3D point clouds from surveys.

### Hpc Type

This recursive type is mainly used for geometric object definition, including the hierarchical values generated by the `STRUCT` function, and interactive graphics visualization on the display device.

An object of `Hpc` type has three fields: a multidimensional matrix `T::MatrixNd`; a vector `childs` (i.e., children) either of elements `Hpc` or of elements `Geometry`, and a `Properties` field of dictionary type, i.e., `Dict{Any, Any}`.

The `mutable struct Hpc` is a typical recursive data structure to represent dynamically a data object of tree type, where the `children` nodes of a node may be in any number since stored into a Julia's `Vector`.

```
mutable struct Hpc
T::MatrixNd
childs::Union{Vector{Hpc}, Vector{Geometry}}
properties::Dict{Any, Any}
# constructor
   function Hpc(T::MatrixNd=MatrixNd(0), childs:: Union{Vector{
    Hpc}, Vector{Geometry}}=[], properties=Dict())
      self = new()
      self.childs = childs
      self.properties = properties
      if length(childs) > 0
         Tdim = maximum([dim(child) for child in childs]) + 1
         self.T = embed(T, Tdim)
      else
         self.T = T
      end
      return self
   end
end
```

## Lar Type

The `mutable struct Lar` is used to represent synthetically a generic cellular or chain complex, together with some of its properties. Given `obj::Lar`, it represents with `obj.d`, `obj.m`, `obj.n`, `obj.V`, and `obj.C`, respectively, the intrinsic dimension (1 for curves, 2 for surfaces, 3 for solids), the number of its coordinates, the number of vertices, and a dictionary of of chain bases or chain operators, stored when already available throughout a computation.

```
mutable struct Lar
   d::Int # intrinsic dimension
   m::Int # embedding dimension (rows of V)
   n::Int # number of vertices  (columns of V)
   V::Matrix{Float64} # object geometry
   C::Dict{Symbol, AbstractArray} # object topology (C for cells
    )
   # inner constructors
   Lar() = new( -1, 0, 0, Matrix{Float64}(undef,0,0), Dict{
    Symbol, AbstractArray}() )
   Lar(m::Int,n::Int) = new( m,m,n, Matrix(undef,m,n), Dict{
    Symbol,AbstractArray}() )
   Lar(d::Int,m::Int,n::Int) = new( d,m,n, Matrix(undef,m,n),
    Dict{Symbol,AbstractArray}() )
   Lar(V::Matrix) = begin m, n = size(V); new( m,m,n, V, Dict{
    Symbol,AbstractArray}() ) end
   Lar(V::Matrix,C::Dict) = begin m,n = size(V); new( m,m,n, V,
    C )  end
   Lar(d::Int,V::Matrix,C::Dict) = begin m,n = size(V); new( d,m
    ,n, V, C )  end
   Lar(d,m,n, V,C) = new( d,m,n, V,C )
end
```

## Geo type

A `mutable struct Geometry` is used as a container for single whole geometric objects, allowing to store the various dimensional cellular subcomplexes that partition the geometric value. Conversely, any hierarchical assembly is stored in `Plasm` within a mixture of `Hpc` and `Geometry` nodes. The `Geometry` data structure contains arrays of integers, denoting the ordered bases of the topological chains of different dimensions that decompose the represented geometric value. It also contains a numeric `db` (data base), implemented as a

Julia dictionary with key the (suitably rounded) coordinate vector sof vertex `point` and with value the corresponding integer index.

```
mutable struct Geometry
   db::Dict{Vector{Float64}, Int}
   points::Vector{Vector{Float64}}
   edges::Vector{Vector{Int}}
   faces::Vector{Vector{Int}}
   hulls::Vector{Vector{Int}}
   # constructor
   function Geometry()
   self = new(
      Dict{Vector{Float64}, Int}(),
      Vector{Vector{Float64}}(),
      Vector{Vector{Int}}(),
      Vector{Vector{Int}}(),
      Vector{Vector{Int}}(),
   )
   return self
   end
end
```

### Topological types

Some abstract types are defined in `Plasm` in order to characterize and document the type of variables and/or parameters within complicated definitions and function codes. They are mainly used within the structures of `Lar` type, to document the topology, and are defined as global `const` symbols.

```
const Points = Matrix{number}
const Cells = Vector{Vector{Int}}
const Cell = SparseVector{Int8, Int}
const Chain = SparseVector{Int8,Int}
const ChainOp = SparseMatrixCSC{Int8,Int}
const ChainComplex = Vector{ChainOp}
```

The `Cells` type stores the cellular bases and some subsets of cells as `Vector` of `Vector` of integers. The `Cell` type is utilized to memorize a single cell's (sparse) representation. The `Chain` type item equals the cell type, and is used only for documentation aims. The `ChainOp` type allows the storage of the topological operators, including boundary and coboundary, and other higher degree operators, e.g., `FV`. The `ChainComplex` type is employed as the multidimensional `Vector` store of chain complexes, by now only in 2D and 3D, with two and three `ChainOp` sparse matrices, respectively.

***Coding 4.1.1 (3-cube topology is a ChainOp object)*** Whereas the expression `cube.C[:FV]` returns a dictionary value of type `Cells`, which contains the basis of our cubic cellular complex, `top` is of `ChainOp` type:

```
cube = LAR(CUBE(3))          #=
Lar(3, 3, 8, [3.0 0.0 … 3.0 0.0; 3.0 3.0 … 3.0 3.0; 0.0 0.0 …
    3.0 3.0], Dict{Symbol, AbstractArray}(:CV => [[1, 2, 3, 4,
    5, 6, 7, 8]], :FV => [[1, 2, 3, 4], [3, 4, 5, 6], [1, 3, 5,
    7], [2, 4, 6, 8], [1, 2, 7, 8], [5, 6, 7, 8]], :EV => [[3,
    4], [2, 4], [1, 2], [1, 3], [5, 6], [4, 6], [3, 5], [5, 7],
    [1, 7], [6, 8], [2, 8], [7, 8]])) =#

FV = cube.C[:FV]             #=
6-element Vector{Vector{Int64}}:
 [1, 2, 3, 4]
 [3, 4, 5, 6]
 [1, 3, 5, 7]
 [2, 4, 6, 8]
 [1, 2, 7, 8]
 [5, 6, 7, 8]

KFV = lar2cop(FV)            #=
6×8 SparseArrays.SparseMatrixCSC{Int8, Int64} with 24 stored
    entries:
 1  1  1  1  ⋅  ⋅  ⋅  ⋅
 ⋅  ⋅  1  1  1  1  ⋅  ⋅
 1  ⋅  1  ⋅  1  ⋅  1  ⋅
 ⋅  1  ⋅  1  ⋅  1  ⋅  1
 1  1  ⋅  ⋅  ⋅  ⋅  1  1
 ⋅  ⋅  ⋅  ⋅  1  1  1  1         =#
```

Of course, `(typeof(FV)==Cells) && (typeof(KFV)==ChainOp) # => true`
. □

*Remark 4.1 (About sparsity)* While in this small case, the `FV` matrix is not very sparse in this small case, the sparsity overgrows as the cellular complex proliferates since the non-zero elements grow linearly with the number of cells. In contrast, the zero elements grow quadratically (with the matrix elements).

## 4.2 Plasm parametric primitives

In this section, we introduce and exemplify several features of the working of `Plasm` with geometric objects. This system is quite different from most geometric and graphics systems since it is based on `FL`-style combinators.

### 4.2.1 Geometric Transformations

The user interface to affine coordinate transformation of geometric objects is given through standard Julia functions and matrices. Internally, `Plasm` implements such mapping of local coordinates using its own multidimensional matrix type, called `MatrixNd`, and the use of the field `T::MatrixNd` within the recursive datatype `Hpc`.

**Definition 4.1 (Geometric transformation)** A *geometric transformation* is a bijective function, i.e., a one-to-one (injective) and onto (surjective) mapping $\mathbb{E}^d \to \mathbb{E}^d$.

By definition, geometric transformations of plane or space are invertible, and hence represented by invertible square matrices. We will see that *rotation*, *scaling*, and *shearing* are linear transformations; *translation* is affine.

#### Homogeneous Coordinates

In computer graphics the *homogeneous coordinates* are often used instead of Cartesian coordinates. In the homogeneous plane or space, lines are mapped to lines, but parallel lines are not conserved parallel. The main reason for this change is the ability to treat affine maps (translation) as linear, and combine smoothly with linear maps (rotation, scaling, etc.)

In homogeneous coordinates the Euclidean plane $\mathbb{E}^2 \setminus \{0\}$ is considered in bijective correspondence with the bundle of lines in $\mathbb{E}^3 \setminus \mathbb{E}^2$ (a model for the projective plane) so that each point $(x, y) \in \mathbb{E}^2$ corresponds to a line $\lambda(W, X, Y)$ such that $(x, y) \equiv \frac{W}{W}, \frac{X}{W}, \frac{Y}{W}) = (1, x, y)$. Same for each $\mathbb{E}^d$, $d \geq 2$. After the division, homogeneous coordinates are said *normalized*.

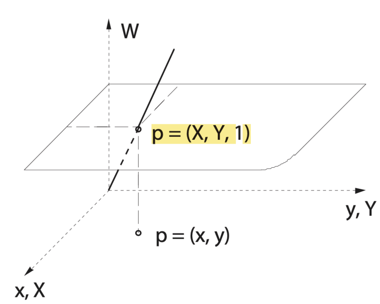



**Fig. 4.1** The homogeneous plane is a model of a projective plane, where all finite points have a homogeneous coordinate equal to one, and the points at infinity have it equal to zero. All the points at infinity form the line at infinity, and all the lines at infinity form the plane at infinity.

In `Plasm`, by design choice to make the multidimensional approach to geometric design more accessible, the added homogeneous coordinate is the first, not the last, as we may see in many computer graphics books.

Even more, for the sake of clarity we can use the `HOMO` operator to transform a $d \times d$ matrix in a $(d+1) \times (d+1)$ matrix, i.e., a $3 \times 3$ on the 2D plane and $4 \times 4$ on 3D space. The type of returned matrix is `MatrixNd`, which is used for dimension-independent programming.

Homogeneous coordinates allow to combine linearly all transformations, using products of their matrices in homogeneous coordinates. In the remainder of this section we describe the geometric effect of each transformation and the structure of the corresponding matrices.

*Remark 4.2* The reader should note that our maps or transformations are invertible functions of a space into itself (automorphisms), represented (even translation, as we will see) by square matrices, i.e. are *rank 2* tensors. Since they are also `Plasm` functions, can be *applied* to geometric objects; as matrices, they multiply the object coordinates.

## 2D rotation

In a *planar rotation* all points of the 2D plane move along an arc of circle, with same angle at center, while the center is the only fixed point. In a *space rotation* there is a straight line of fixed points (the axis) passing for the origin. All the other 3D points describe a circle arc with the same angle along the plane (orthogonal to rotation axis) which they belong to.

Let us show (see Figure **??**) how unit vectors $e_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $e_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, columns of the matrix $\begin{pmatrix} e_1 & e_2 \end{pmatrix}$ are transformed by the (yet unknown) $R(\alpha)$ rotation matrix into the columns of the matrix at right-hand side:

$$\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} = R(\alpha) \begin{pmatrix} e_1 & e_2 \end{pmatrix}. \quad \text{Hence we have:} \quad R(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

There is only one class of planar rotations, parameterized by $\alpha$, the *rotation angle* about the origin. Conversely, we will see three classes of elementary space rotations, parameterized by $\alpha_x, \alpha_y, \alpha_z$, the rotation angles about each coordinate axis.

***Coding 4.2.1 (Plasm notation for rotation)*** The plane rotation function in `Plasm` is: `R([1,2])(α)` because its effect is to change the first and second coordinates of the 2D model it is applied to. They are applied to a planar geometric object of `Hpc` type, by using the `STRUCT` operator (see Section **??**) that contains `Hpc` values and transformation tensors:

```
SQUARE(d) = CUBOID([d,d])        #=
SQUARE (generic function with 1 method)     =#

obj = R(1,2)(π/4)(SQUARE(1))         #=
```

```
Hpc(MatrixNd([[1.0, 0.0, 0.0], [0.0, 0.7071067811865476,
    -0.7071067811865475], [0.0, 0.7071067811865475,
    0.7071067811865476]]), Hpc(MatrixNd(3), Hpc(MatrixNd(3),
    Geometry([[0.0, 0.0], [1.0, 0.0], [1.0, 1.0], [0.0, 1.0]],
    hulls=[[1, 2, 3, 4]])))) =#

VIEW(obj)
```

*Remark 4.3* `CUBOID(shape)::Hpc` is the generator of multidimensional hyper-parallelopipeds, depending on length and content of `shape` vector.

`CUBOID([1,1])` is the unit square; `CUBOID([1,2,3])` is the parallelopiped of sides 1, 2, and 3; `CUBOID([1,1,1,1])` is the 4D unit hypercube.

## Elementary rotations

The multidimensional `Plasm` language has the following definition of elementary rotation, that allows to rotate a $r$-model ($r \leq d$) in any dimension $d \geq 2$.

**Definition 4.2 (Elementary rotation)** The reader should note that the *elementary* rotation is defined in any dimension $d$ such that only 2 coordinates are changed by the rotation.

*Remark 4.4* It is easy to see that in any dimension $d$ there are `binomial`$(d,2)$ elementary rotations, how many are the ways to choose 2 coordinates over $d$. Hence we have 1 for $d = 2$, 3 for $d = 3$, 6 for $d = 3$, and so on.

Assume that the rotation axes are $e_1, e_2, e_3$, with rotation angles $\alpha, \beta, \gamma$ respectively. The corresponding elementary matrices, derivable as before by change of coordinates, are:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}, \; R_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{pmatrix}, \; R_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 \\ \sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4.1}$$

In `Plasm` the elementary rotations are represented, respectively, by the tensors: `R([2,3])`$(\alpha)$, `R([1,3])`$(\beta)$, and `R([1,2])`$(\gamma)$.

***Coding 4.2.2 (Elementary rotation)*** We use an interesting 3D polyhedron, called permutahedron, to show the application of the tensor `R([1,2])(pi/2)` to it:

```
obj = R([1,2])(pi/2)( PERMUTAHEDRON(3) );
VIEW( obj )
```

*Remark 4.5 (Reduction of visual noice)* Just note that in all `Plasm` geometric operators, the constraint of using functions as unary has been relaxed, in order to make possible to write, e.g., `obj = R(1,2)(pi/2)(obj)` instead than `obj = R([1,2])(pi/2)(obj)`. In the remainder we use always this new style.

***Coding 4.2.3 (Permutahedron)*** The reader might be curious to see how such important and beautiful polyhedron [38] whose vertex coordinates are the permutations of the first $d$ natural numbers. Iy is generated in `Plasm`:

```
function PERMUTAHEDRON(d)
    vertices = ToFloat64(PERMUTATIONS(collect(1:d+1)))
    center = MEANPOINT(vertices)
    cells = [collect(1:length(vertices))]
    object = MKPOL(vertices, cells, [[1]])
    object = T(INTSTO(d))(-center)(object)
    for i in 1:d
        object = R(i,d+1)(pi/4)(object)
    end
    object = PROJECT(1)(object)
    return object
end
```

The `Plasm` function `INTSTO(d)` (integers to $d$) is used to generate the sequence `[1,2,...,d]`, extremes included. The other functions are easy to understand.□

## General rotation in 3D

A rotation of 3D space has a fixed line of points (the rotation axis) passing through the origin. We may compute the corresponding matrix as a function of a direction vector for the axis and a real value for the rotation angle. For this purpose we can compone three linear transformations by multiplication of their matrices. Therefore we have:

**Definition 4.3 (General 3D rotation with axis $d$ and angle $\alpha$)** Clearly, the ordering of transformations is from right to left:

$$R(d,\alpha) = Q^{-1}(d)\, R_z(\alpha)\, Q(d)$$

First, a space rotation that brings the vector $d$ on a coordinate axis, say $e_3$; second, a space rotation $R_z(\alpha)$ about the $z$-axis; third, the inverse of the first transformation, so to bring the rotation axis in its original direction.

$Q(d)$ must transform the unit vector $d$ to the $e_3$ unit vector. So, we may compute the coordinate transformation that brings three orthonormal vectors $(u_1, u_2, u_3)$ to become the standard basis $(e_1, e_2, e_3)$. We can choose the triple:

$$\begin{aligned} u_3 &= d / \|d\|, \\ u_2 &= (u_3 \times e_3) / \|u_3 \times e_3\|, \\ u_1 &= u_2 \times u_3, \end{aligned} \tag{4.2}$$

to write the transformation of coordinates:

$$(e_1, e_2, e_3) = Q(d)\,(u_1, u_2, u_3)$$

so that we have:

$$Q(d) = (u_1, u_2, u_3)^{-1}$$

But $Q(d)$ maps orthonormal vectors to orthonormal vectors, hence it is a normal transformation, so its inverse is equal to its transpose. So, we can write:

$$R(d, \alpha) = Q^t(d)\,R_z(\alpha)\,Q(d). \tag{4.3}$$

***Coding 4.2.4 (3D General Rotation matrix)*** Let's use a test-driven programming style, with parameter values easy to test Rotate of 45 degrees about the diagonal axis the unit cube with a vertex on the origin, i.e. the model generated by the `CUBE(1)` expression in `Plasm`. We may follow this procedure using a functional approach:

```
using Plasm, LinearAlgebra
d = [1,1,1];
u₃ = normalize(d);
u₂ = normalize(u₃ × [0,0,1]);
u₁ = u₂ × u₃;
```

and write the following matrix for the transformation of coordinates that maps the $e_3$ axis to the direction of the `d` vector.

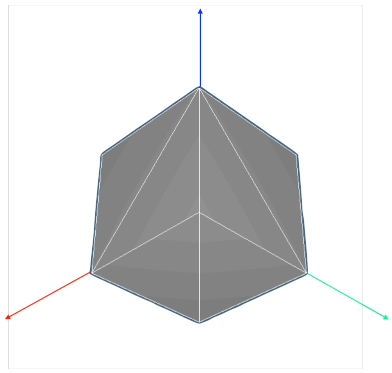
```
Q(d) = [u₁ u₂ u₃]'
```

The single quote stands for the Julia `transpose` of a matrix. □

***Coding 4.2.5 (General 3D rotation tensor)*** In what follows, `MAT` transforms a Julia `Matrix` into a `Plasm` tensor applicable to `Hpc` values. The `HOMO` function apply to a square matrix, adding new unitary first row and column, for homogeneous coordinates (see Section **??**).

```
GR(d,α) = MAT(HOMO(Q(d)')) ∘ R(1,2)(α) ∘ MAT(HOMO(Q(d)))
```

The `GR` (general rotation) is a `Plasm` tensor depending on the axis `d` and the angle `α`. Our geometric model is therefore rotated and viewed as follows. □



```
rotated = GR([1,1,1],π/3)(CUBE(1))
VIEW(rotated)
```

## Scaling

In a scaling transformation, all points are moved along the line passing for the origin they belong to. The scaling is said *elementary* when only one of the coordinates changes. There are two scaling parameters $s_x, s_y$ in 2D geometry and three scaling parameters $s_x, s_y, s_z$ in 3D, to be used in scalar products by the point coordinates. The transformation can be a *dilatation* of space when scaling parameters are greter than one, or a *contraction* of space when scaling parameters are lesser than one.

$$S(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{pmatrix}, \; S_x = \begin{pmatrix} s_x & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; S_y = \begin{pmatrix} 1 & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}, \; S_z = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & s_z \end{pmatrix}$$

The scaling matrices are diagonal. The origin remains fixed. In fact:

$$S \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^t = \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}^t.$$

Hence, a scaling transformation is linear. It is easy to see that scale transformations are multiplicative, commutative, and associative because the matrix is diagonal:

$$S(s_x, s_y, s_z) = S_x(s_x)\, S_y(s_y)\, S_z(s_z)$$

.

***Coding 4.2.6 (How to scale a Plasm model?)*** As in the previous coding example, let's go to use the `cube(1)` as our model object.

```
scaledcube1 = S(1,2,3)(.1,.1,10)(CUBE(1))
scaledcube2 = S(3)(100)(CUBE(1))
```

***Coding 4.2.7 (How to scale a Plasm model?)*** Note that the effect of transformations impacts only the homogeneous matrices ahead of `Hpc` values.

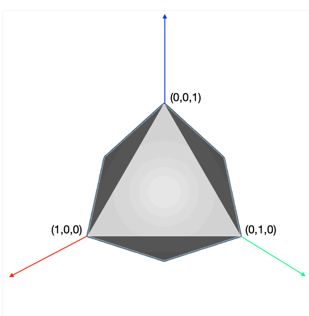
```
scaledcube1 = S(1,2,3)(.1,.1,10)(CUBE(1))   #=
Hpc(MatrixNd([[1.0, 0.0, 0.0, 0.0], [0.0, 0.1, 0.0, 0.0], [0.0,
    0.0, 0.1, 0.0], [0.0, 0.0, 0.0, 10.0]]), Hpc(MatrixNd(4),
    Hpc(MatrixNd(4), Geometry([[0.0, 0.0, 0.0], [1.0, 0.0, 0.0],
     [0.0, 1.0, 0.0], [1.0, 1.0, 0.0], [0.0, 0.0, 1.0], [1.0,
    0.0, 1.0], [0.0, 1.0, 1.0], [1.0, 1.0, 1.0]], hulls=[[1, 2,
    3, 4, 5, 6, 7, 8]])))) =#
scaledcube2 = S(2)(100)(SQUARE(1))  #=
Hpc(MatrixNd([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0,
    100.0]]), Hpc(MatrixNd(3), Hpc(MatrixNd(3), Geometry([[0.0,
    0.0], [1.0, 0.0], [1.0, 1.0], [0.0, 1.0]], hulls=[[1, 2, 3,
    4]])))) =#
```

Of course, `S(1,2,3)(.1,.1,10)` and `S(2)(100)` are tensor objects. □

***Coding 4.2.8 (Construction of octahedron model)*** As an exciting coding example, we show a simple construction of an octahedron model, starting from the 3D SIMPLEX model.

```
tetra = SIMPLEX(3);
twotetra = STRUCT( tetra, S(1)(-1), tetra );
fourtetra = STRUCT( twotetra, S(2)(-1), twotetra );
octahedron = STRUCT( fourtetra, S(3)(-1), fourtetra );
```

Looking at the whole cellular complex corresponding to the solid model `octahedron::Hpc` is worthwhile. For this purpose, we transform it into an object of `Lar` type:




```
VIEW(octahedron::Hpc)
```

**Fig. 4.2** Plasm viewing generator expression. Remember that VIEW applies to Hpc values.

***Coding 4.2.9 (The cellular complex)*** Let's note that the `octahedron::Hpc` is viewed, and that the `octahedron::Lar` is explored for stored data:

```
LAR(Octahedron).V    #=
3×7 Matrix{Float64}:
  0.0  -1.0  0.0   0.0  1.0  0.0  0.0
 -1.0   0.0  0.0   0.0  0.0  1.0  0.0
  0.0   0.0  0.0  -1.0  0.0  0.0  1.0    =#
LAR(Octahedron).C    #=
Dict{Symbol, AbstractArray} with 3 entries:
  :CV => [[1, 2, 3, 4], [1, 3, 4, 5], [2, 3, 4, 6], [3, 4, 5,…
  :FV => [[1, 2, 3], [2, 3, 4], [1, 3, 4], [1, 2, 4], [1, 3, …
  :EV => [[2, 3], [1, 3], [1, 2], [3, 4], [2, 4], [1, 4], [3,…=#
```

For `#C, #F, #E, #V`, we see, looking at `.V` and `.C` above:

```
AA(LEN)(values(octahedron.C))'  #=
1×3 adjoint(::Vector{Int64}) with eltype Int64:
 8  20  18      =#
```

Therefore, we may see that the combinatorial (simplicial) complex corresponding to the 3D octahedron is made by $8 + 20 + 18 + 7 = 53$ cells of dimension 3, 2, 1, and 0, respectively (see Figure 4.2). □

## Shearing

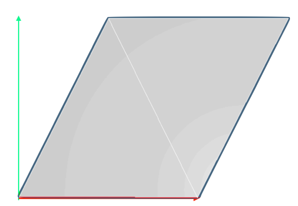
In a 2D elementary *shearing tranformation* all points of each line (plane in 3D) orthogonal to a coordinate axis move by summing one (fixed) vector. The coordinate line (plane in 3D) remain fixed, and the translation vector change linearly with the distance of its line (plane) from the origin. Each of two elementary planar shearing depends on a single scalar parameter (the translation of the line at unit distance from the coordinate line).

Each of the three elementary space shearing in 3D conversely depends on two scalar parameters (the coordinates of the planar translation vector of the plane at unit distance from the coordinate plane. Their 2D and 3D matrices are as follows.

$$H_x(h_x) = \begin{pmatrix} 1 & 0 \\ h_x & 1 \end{pmatrix}, \qquad H_y(h_y) = \begin{pmatrix} 1 & h_y \\ 0 & 1 \end{pmatrix};$$

$$H_x(h_y, h_z) = \begin{pmatrix} 1 & 0 & 0 \\ h_y & 1 & 0 \\ h_z & 0 & 1 \end{pmatrix}, \ H_y(h_x, h_z) = \begin{pmatrix} 1 & h_x & 0 \\ 0 & 1 & 0 \\ 0 & h_z & 1 \end{pmatrix}, \ H_z(h_x, h_y) = \begin{pmatrix} 1 & 0 & h_x \\ 0 & 1 & h_y \\ 0 & 0 & 1 \end{pmatrix}.$$

An elementary shearing differs from the identity matrix only for the elements of a single column, both in 2D and in 3D, and also in homogeneous 4D coordinates. In `Plasm`, the shearing tensor is named `H` and has the following semantics: first, indicate the column index; then give the $d-1$ ordered transformation parameters, i.e., one in 2D and two in 3D, some of which possibly zeros. Therefore we have `H(col)(pars)`.

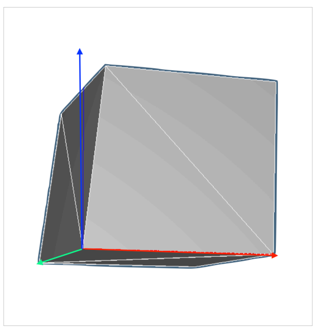


```
SQUARE(d) = CUBOID([d,d])
shearedsquare = H(2)(.5)(SQUARE(1))

VIEW(shearedsquare)
```

**Fig. 4.3** Unit square sheared on the (second) coordinate $y$. The $y$ of model points does not change.

Typically, `shearing` is used by typesetting systems of computerized typography to get *italic* versions of character fonts. Note also above how we define a parametric square (with a vertex on the origin).



```
shearedcube = H(3)(.2,.3)(CUBE(1))

VIEW(shearedcube)
```

**Fig. 4.4** Unit cube sheared on the (third) coordinate $z$. The $z$ of points do not change.

It is worthwhile to remark that the `H` mapping, as `R`, `GR`, `S`, `T`, `MAT`, and `HOMO` are dimension-independent, so can be applied to models of whatever embedding dimension $d$ of geometric models. Homogeneous *normalized* matrices are used for implementation purpose.

## Translation

**Definition 4.4 (Translation transformation)** a translation is an invertible transformation of Euclidean space $\mathbb{E}^d$ generated by summing a fixed vector to all points.

A translation of plane $\mathbb{E}^2$ (or space $\mathbb{E}^3$) is not a linear transformation, since it moves the origin, but it is an *affine* transformation, since all $\mathbb{E}^2$ (or $\mathbb{E}^3$) mapped points change by sum with a fixed vector (an *affine action*). □

Hence, a 2D or 3D translation depends on two (or three) scalar parameters, i.e., by the coordinates of the *translation vector*. We may therefore translate, using coordinates, a generic vector $v = (v_i) \in \mathbb{E}^d$:

$$v' = v + t \tag{4.4}$$

where $t = (t_i)$ is the translation vector, applied to all points in $\mathbb{E}^d$.

### Translation in homogeneous coordinates

The translation 4.4 is reduced to a linear transformation and hence is representable by a product with a square matrix when using normalized homogeneous coordinates. Let's remind our choice to use as homogeneous the first coordinate. For example, a translation of $\mathbb{E}^3$ is representable as

$$T(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ t_x & 1 & 0 & 0 \\ t_y & 0 & 1 & 0 \\ t_z & 0 & 0 & 1 \end{pmatrix} \tag{4.5}$$

We can see the equivalence between translation with Cartesian coordinates and homogenous normalized (affine) coordinates. Let $v = (x, y, z)$ be a point in 3D Euclidean space $\mathbb{E}^3$, and $v' = (w = 1, x, y, z)$ the same point in $\mathbb{R}^4$:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ t_x & 1 & 0 & 0 \\ t_y & 0 & 1 & 0 \\ t_z & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 \\ x + t_x \\ y + t_y \\ z + t_z \end{pmatrix}$$

Just notice that a translation in 3D is actually a shearing $H_w(t_x, t_y, t_z)$ orthogonal to the added component in normalized homogeneous coordinates.

***Coding 4.2.10 (Translation of 3D geometric object)*** In `Plasm` we translate a geometric object of `Hpc` type via tensor application:

```
t_cube1 = T(1,2,3)(.5,.5,.5)(CUBE(1))
t_cube2 = T(3)(1)(CUBE(1))

VIEW(t_cube1);  VIEW(t_cube2)
```

A triple application of `T` function is needed: first to indices, then to translation paramenters, finally to the object of `Hpc` type to be translated. □

***Coding 4.2.11 (Parametric linear ladder stair)*** The `step` is a `Hpc` solid obtained by product of three line segments of given sizes. An array of $n$ pairs `[ move, step]` is generated and concatenated by the `CAT` operator. Finally, the semantics of STRUCT (see Section 4.3) produces the whole parametric object, shown in Figure 4.5.

```
function Ladder(lx,ly,lz, n)::Hpc
    step   = QUOTE(lx) * QUOTE(ly) * QUOTE(lz)
    move   = T(2,3)(0.8*ly, 0.8*lz)
    ramp   = STRUCT( CAT([[step, move] for k=1:n]) )
end #=
Ladder (generic function with 1 method) =#
stair = Ladder(.8, .22, .18, 15);

VIEW(stair)
```

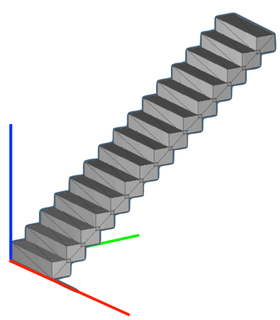


**Fig. 4.5** Simple linear scale, demostrating an iterative use of tensors in STRUCT. Of course, not only the number, but also the size and even the shape of step model can be parametrized, as arguments of a geometric function returning Hpc objects.

## 4.3 Hierarchical assembly of geometric objects

## 4.4 Attach properties to geometry

## 4.5 Design documentation (Jupyter notebooks)

## 4.6 Export geometry

## References


1. Arakaki., T.: Julia Data Parallel Computing. URL `https://juliafolds.github.io/data-parallelism/tutorials/quick-introduction/`. [retrieved june 27, 2023]
2. Arnold, D.N.: Finite Element Exterior Calculus, *CBMS-NSF Regional Conference Series in Applied Mathematics*, vol. 93. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA (2018)
3. Aubanel, E.: Elements of Parallel Computing. Chapman Hall/CRC Press (2016)
4. Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Commun. ACM **21**(8), 613641 (1978). DOI 10.1145/359576.359579. URL `https://doi.org/10.1145/359576.359579`
5. Backus, J., Williams, J., Wimmers, E.: An introduction to the programming language FL. In: D. Turner (ed.) Research Topics in Functional Programming. Addison-Wesley, Reading, MA (1990)
6. Backus, J., Williams, J.H., Wimmers, E.L., Lucas, P., Aiken, A.: FL LANGUAGE MANUAL. PARTS l AND 2. Tech. Rep. RJ 7100 (67163), IBM Almaden Research Center (1989)
7. Bezanson, J., Edelman, A., Karpinski, S., Shah, V.B.: Julia: A fresh approach to numerical computing. SIAM Review **59**(1), 65–98 (2017). URL `https://doi.org/10.1137/141000671`
8. Carlsson, C.: Syntax highlighting – OhMyREPL. URL `https://kristofferc.github.io/OhMyREPL.jl/latest/`. [retrieved may 22, 2024]
9. Cimrman, R.: Sparse matrices in scipy. In: G. Varoquaux, E. Gouillart, O. Vahtras, P. deBuyl (eds.) Scipy lecture notes, release: 2022.1 edn., p. Section 2.5. Zenodo (2015). DOI 10.5281/zenodo.594102. URL `https://scipy-lectures.org/advanced/scipy_sparse/index.html`
10. Danisc, S., Kavalar, M., Dombrowski, M., Markovics, P.: An Introduction to GPU Programming in Julia. URL `https://nextjournal.com/sdanisch/julia-gpu-programming`. [retrieved june 29, 2023]
11. Delfinado, C., Edelsbrunner, H.: An incremental algorithm for betti numbers of sinplicial complexes on the 3-sphere. Computer Aided Geometric Design **12**, 771–784 (1995)
12. DiCarlo, A., Milicchio, F., Paoluzzi, A., Shapiro, V.: Chain-based representations for solid and physical modeling. Automation Science and Engineering, IEEE Transactions on **6**(3), 454 –467 (2009). DOI 10.1109/giorgio
13. DiCarlo, A., Paoluzzi, A., Shapiro, V.: Linear algebraic representation for topological structures. Computer-Aided Design **46**, 269–274 (2014). DOI 10.1016/j.cad.2013.08.044. URL `https://doi.org/10.1016/j.cad.2013.08.044`
14. DiCarlo, A., Paoluzzi, A., Shapiro, V.: Linear algebraic representation for topological structures. Comput. Aided Des. **46**, 269–274 (2014). DOI 10.1016/j.cad.2013.08.044. URL `http://dx.doi.org/10.1016/j.cad.2013.08.044`
15. Ferrucci, V.: Generalised extrusion of polyhedra. In: Proceedings on the Second ACM Symposium on Solid Modeling and Applications, SMA '93, p. 3542. Association for Computing Machinery, New York, NY, USA (1993). DOI 10.1145/164360.164376. URL `https://doi.org/10.1145/164360.164376`
16. Ferrucci, V., Paoluzzi, A.: Extrusion and boundary evaluation for multidimensional polyhedra. Comput. Aided Des. **23**(1), 4050 (1991). DOI 10.1016/0010-4485(91)90080-G. URL `https://doi.org/10.1016/0010-4485(91)90080-G`
17. Hatcher, A.: Algebraic topology. Cambridge University Press (2002)
18. Julia: Manual: Base/lib-collections/iteration. URL `https://docs.julialang.org/en/v1/base/collections/#lib-collections-iteration`. [retrieved june 25, 2023]

19. Julia: Manual: Distributed-Computing. URL https://docs.julialang.org/en/v1/manual/distributed-computing/#Multi-processing-and-Distributed-Computing. [retrieved june 29, 2023]
20. Julia: Manual: Linear Algebra. URL https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#man-linalg. [retrieved july 3, 2023]
21. Julia: Manual: LinearAlgebra.factorize. URL https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/#LinearAlgebra.factorize/. [retrieved june 25, 2023]
22. Julia: Manual: Metaprogramming. URL https://docs.julialang.org/en/v1/manual/metaprogramming/. [retrieved june 21, 2023]
23. Julia: Manual: Modules. URL https://docs.julialang.org/en/v1/manual/modules/#modules. [retrieved june 29, 2023]
24. Julia: Manual: Multi-Threading. URL https://docs.julialang.org/en/v1/manual/multi-threading/#man-multithreading. [retrieved june 29, 2023]
25. Julia: Manual: Parallel Computing. URL https://docs.julialang.org/en/v1/manual/parallel-computing/#Parallel-Computing. [retrieved june 26, 2023]
26. Julia: Manual: Parallel Computing. URL https://docs.julialang.org/en/v1/base/parallel/#Tasks. [retrieved june 27, 2023]
27. Julia: Manual: stdlib/SparseArrays. URL https://docs.julialang.org/en/v1/stdlib/SparseArrays/#man-csc/. [retrieved june 25, 2023]
28. Julia: Manual: Types. URL https://docs.julialang.org/en/v1/manual/types/#man-types. [retrieved june 30, 2023]
29. Julia Community: How to develop a Julia package. URL https://julialang.org/contribute/developing_package/. [retrieved june 29, 2023]
30. JuliaGPU: A gentle introduction to parallelization and GPU programming in Julia. URL https://cuda.juliagpu.org/stable/tutorials/introduction/#Introduction. [retrieved june 29, 2023]
31. Kamiski, B.: Julia for Data Analysis. Manning (2023)
32. Mainon, P.: Writing type-stable julia code (2021). URL https://blog.sintef.com/industry-en/writing-type-stable-julia-code/
33. Paoluzzi, A.: Geometric Programming for Computer Aided Design. John Wiley Sons, Chichester, UK (2003). URL https://onlinelibrary.wiley.com/doi/book/10.1002/0470013885
34. Paoluzzi, A., Pascucci, V., Vicentino, M.: Geometric programming: a programming approach to geometric design. ACM Trans. Graph. **14**(3), 266–306 (1995). DOI 10.1145/212332.212349. URL http://doi.acm.org/10.1145/212332.212349
35. Paoluzzi, A., Scorzelli, G., Vicentino, M.: Securing the cultural heritage via geometric programming and modeling. Tech. rep., Dept of Computer Science and Engineering, Roma Tre University (2009). URL https://www.academia.edu/47017676
36. Paoluzzi, A., Shapiro, V., DiCarlo, A., Furiani, F., Martella, G., Scorzelli, G.: Topological computing of arrangements with (co)chains. ACM Trans. Spatial Algorithms Syst. **7**(1) (2020). DOI 10.1145/3401988. URL https://doi.org/10.1145/3401988
37. Paoluzzi, A., Shapiro, V., DiCarlo, A., Scorzelli, G., Onofri, E.: Finite algebras for solid modeling using julias sparse arrays. Computer-Aided Design **155**, 103436 (2023). DOI https://doi.org/10.1016/j.cad.2022.103436. URL https://www.sciencedirect.com/science/article/pii/S0010448522001695
38. Permutohedron: Permutohedron — Wikipedia, the free encyclopedia (2023). URL https://en.wikipedia.org/wiki/Permutohedron. [Online; accessed 31-May-2024]
39. Roth, A., Weisstein, E.W.: Standard basis. In: MathWorld, p. Algebra > Linear Algebra > Linear Systems of Equations. A Wolfram Web Resource (2005). URL https://mathworld.wolfram.com/StandardBasis.html

40. Rowland, T.: Characteristic function. In: From MathWorld, p. Foundations of Mathematics > Set Theory > Sets. A Wolfram Web Resource (2005). URL ttps://mathworld.wolfram.com/CharacteristicFunction.html
41. Scorzelli, G.: Pyplasm library (2023). URL https://libraries.io/pypi/pyplasm
42. Scorzelli, G., Paoluzzi, A.: Plasm.jl: v0.1.0 (2023). URL https://github.com/scrgiorgio/Plasm.jl
43. Weisstein, E.W.: Julia set. From MathWorld–A Wolfram Web Resource URL https://mathworld.wolfram.com/JuliaSet.html
44. Whitaker, S.: Mastering the Julia REPL. URL https://blog.glcs.io/julia-repl#heading-starting-the-julia-repl. [retrieved may 22, 2024]
45. Whitney, H.: Geometric Integration Theory. Princeton University Press, Princeton (1957). DOI doi:10.1515/9781400877577. URL https://doi.org/10.1515/9781400877577
46. Wikibooks: Introducing julia/dictionaries and sets — wikibooks, the free textbook project (2020). URL https://en.wikibooks.org/. [Online; accessed 20-June-2023]
47. Williams, J.H., Wimmers, E.L.: Sacrificing Simplicity for Convenience: Where Do You Draw the Line? In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’88, p. 169179. Association for Computing Machinery, New York, NY, USA (1988). DOI 10.1145/73560.73575. URL https://doi.org/10.1145/73560.73575